STL：函数对象和智能指针（OOP）

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- string字符串类
- iostream输入输出流
- 字符串处理与正则表达式

本讲内容提要

- 函数对象
- 智能指针与引用计数

函数对象

回忆：什么是函数

- 看一个例子
  - 如果flag是1，则对每个元素调用inc；否则调用dec

```
#include <iostream>
int arr[5] = { 5, 2, 3, 1, 7 };
void increase(int &x){x++;}
void decrease(int &x){x--;}
int main()
{
    int flag; std::cin >> flag;
    if (flag == 1) {
        for (int &x : arr) {   increase(x);}
    }else{
        for (int &x : arr) {   decrease(x);}
    }
    return 0;
}
```

  - 仅仅只有调用的函数不同，如何减少重复的逻辑?

回忆：什么是函数

- 函数可以赋值吗？

```
if (flag == 1)
    func = increase;
else
    func = decrease;
for (int &x : arr) { func(x);}
```

- 当然可以。在C中，func是指向函数的指针。

  - `void (*func)(int&);` //函数指针的声明

返回值　　参数列表

指针符号　声明的变量名

回忆：什么是函数

- 函数的类型比较难写，使用auto可以自动推断

```
//auto必须要有初始化
auto func = flag==1?increase:decrease;
for (int &x : arr) { func(x);}
```

  - auto自动推断出func的类型为void (*)(int&);

- 和数组类似：
  - 数组名 = 指向数组第一个元素的指针
  - 函数名 = 指向函数的指针

回忆：什么是函数

```
#include <iostream>
using namespace std;
void increase(int &x){x++;}
void decrease(int &x){x--;}
int main()
{
    int flag; cin >>flag;

    int arr[] = {1,2,3,4,5};
    void (*func)(int&); //声明函数指针
    if(flag==1) {func=increase; }
    else {func=decrease;}
    //auto func = flag==1?increase:decrease; //和上两行效果相同
    for (int &x:arr) { func(x); cout << x;}

    return 0;
}
```

函数作为变量

- 例子：给一个长度为n的数组，如何排序？
  - std::sort　来自<algorithm>
  - template<class Iterator>
    void sort (Iterator first, Iterator last);

```
#include <algorithm>
#include <iostream>
using namespace std;
int main(){
    int arr[5] = { 5, 2, 3, 1, 7 };
    std::sort(arr, arr + 5);
    for (int x : arr) {cout << x << " ";}
    // 1 2 3 5 7
    return 0;
}
```

函数作为变量

- 如果想倒转排序?
- 注意到sort还重载了另一套参数

```
template <class Iterator, class Compare>
void sort (Iterator first,
                Iterator last, Compare comp);
```

比较函数comp：

```
bool comp(int a, int b)
{
    return a > b;
    //comp函数传入两个值
    //若a在b前，则返回true
    //若a在b后 或 a等于b，则返回false
}
```

函数作为变量

```
#include <algorithm>
#include <iostream>
using namespace std;

bool comp(int a, int b)
   {  return a > b;  }
int main(){
   int arr[5] = { 5, 2, 3, 1, 7 };
   std::sort(arr, arr + 5, comp);
   for (int x : arr) {
      cout << x << " ";
   }  // 7 5 3 2 1
   return 0;
}
```

函数作为变量

- 实际上，Compare就是comp的类型
  - 函数指针：bool (*)(int,int)

- STL提供了预定义的比较函数(#include <functional>)
  - 从小到大

    sort(arr, arr+5, less<int>())
  - 从大到小

    sort(arr, arr+5, greater<int>())
- 疑问：
  - 对比 sort(arr, arr + 5, comp)
  - greater<int>()为什么带括号？是什么？

函数对象

- 实际上，greater<int>()是一个对象
  - greater是一个模板类
  - greater<int>　用int实例化的类
  - greater<int>()　该类的一个对象
- 同时，它表现的像一个函数

```
#include <functional>
#include <iostream>
using namespace std;
int main() {
    auto func = greater<int>();
    cout << func(2, 1) << endl;   //True
    cout << func(1, 1) << endl;   //False
    cout << func(1, 2) << endl;   //False
    return 0;
}
```

- 因此，这种对象被称为函数对象

如何实现函数对象

- 注意三个const
  - 排序中，comp不能修改数据
  - 一般情况下，comp也不应该修改自身

```
#include <iostream>
using namespace std;
template<class T>
class Greater {
public:
    bool operator()(const T &a, const T &b) const {
        return a > b;
    }
};
int main(){
    auto func = Greater<int>();
    cout << func(2, 1) << endl;   //True
    cout << func(1, 1) << endl;   //False
    cout << func(1, 2) << endl;   //False
    return 0;
}
```

如何实现函数对象

- 函数对象的要求有哪些?
  - 需要重载operator()运算符
  - 并且该函数需要是public访问权限

- 小知识：Duck Typing 鸭子类型，通过对象表现的方法和属性确定对象的适用性
  - 如果一个物体，叫声像鸭子、走路像鸭子，那么它就是鸭子;
  - 如果一个对象，用起来像函数，那么它就是函数对象!

    https://en.wikipedia.org/wiki/Duck_typing
  - C++没有严格定义什么是函数对象
  - 但是实践上按Duck Typing来处理

实现自己的sort

```
bool comp(int a, int b)
{
    return a > b;
}

//sort(arr, arr+5, comp);
```

```
template<class T>
class Greater
{
public:
    bool operator()(const T& a,
     const T& b) const {return a > b;}
};

//sort(arr, arr+5, greater<int>())
```

- sort的第三个参数是什么类型?

```
template <class Iterator, class Compare>
void sort (Iterator first, Iterator last,
                    Compare comp);
```

  - 模板类型，可以接受函数指针/函数对象

实现自己的sort

```
#include <algorithm>
#include <functional>
using namespace std;

bool comp(int a, int b)
{
    return a > b;
}
template<class Iterator, class Compare>
void mysort(Iterator first, Iterator last, Compare comp)
//mysort的时间复杂度为O(n^2)，std::sort的时间复杂度为O(nlogn)
{
    for (auto i = first; i != last; i++)
        for (auto j = i; j != last; j++)
            if (!comp(*i, *j)) swap(*i, *j);
}
int main()
{
    int arr[5] = { 5, 2, 3, 1, 7 };
    mysort(arr, arr + 5, comp);
    mysort(arr, arr + 5, greater<int>());
    //既可接受函数指针，又可接受函数对象
    return 0;
}
```

自定义类型的排序

- 假设有一个class People

```
class People
{
public:
   int age, weight;
};
```

- 如何按照年龄从小到大排序?

自定义类型的排序

- 方法一：重载小于运算符

```
#include <algorithm>
using namespace std;
class People
{
public:
    int age, weight;
    bool operator<(const People &b) const
    {
        return age < b.age;
    }
};
int main() {
    vector<People> vec = {{18, 50}, {16, 40}};
    sort(vec.begin(), vec.end());
    return 0;
}
```

People的operator<
一定按年龄计算吗？
体重怎么办？

自定义类型的排序

- 方法二：定义比较函数

```
#include <algorithm>
using namespace std;

class People
{  public: int age, weight;  };

bool compByAge(const People &a, const People &b)
    {  return a.age < b.age; }

int main() {
    vector<People> vec = {{18, 50}, {16, 40}};
    sort(vec.begin(), vec.end(), compByAge);
    return 0;
}
```

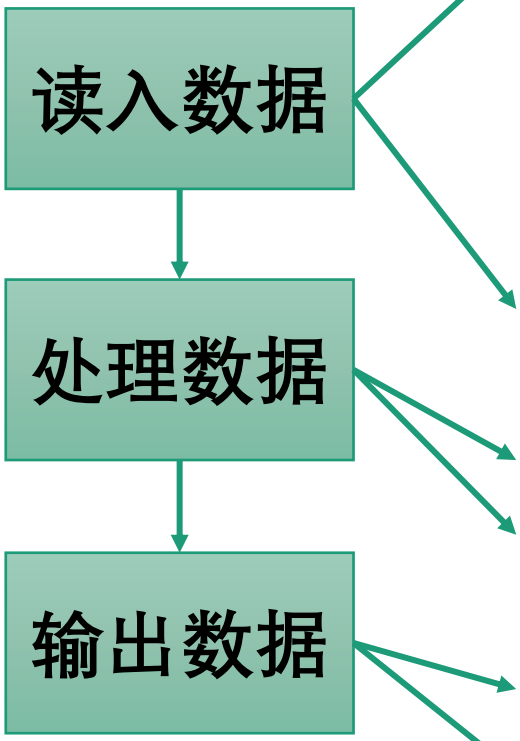
自定义类型的排序

- 方法三：定义比较函数对象

```
#include <algorithm>
using namespace std;
class People
{  public:   int age, weight;  };

class AgeComp {
public:
    bool operator()(const People &a, const People &b) const
        {  return a.age < b.age;  }
};

int main() {
    vector<People> vec = {{18, 50}, {16, 40}};
    sort(vec.begin(), vec.end(), AgeComp());
    return 0;
}
```

例子：一个简单计算器

- 分为三个步骤，每个步骤都有可选的方式




```
string readFromScreen(){
    string input;
    getline(cin, input);
    return input;
}
string readFromFile()

string calculateAdd(string)
string calculateMul(string)

void writeToScreen(string)
void writeToFile(string)
```

例子：一个简单计算器

- 基于虚函数的模板设计模式

```
class CalculatorBase
{
public :
    virtual string read();
    virtual string calculate(string);
    virtual void calculate(string);
    void process()
    {
        string data = read();
        string output = calculate(data);
        write(output);
    }
};
```

见L9课件

例子：一个简单计算器

- 如果使用函数对象？能不能写成

```
void process(ReadFunc read,
             CalFunc calculate,
             WriteFunc write)
{
    string data = read();
    string output = calculate(data);
    write(output);
}
process(readFromScreen,
        calculateAdd, writeToFile);
```

- ReadFunc，CalFunc，WriteFunc分别是什么？

例子：一个简单计算器

- 如果参数只有函数指针
  - ReadFunc = string(*)(void)
  - CalFunc = string(*)(string)
  - WriteFunc = void(*)(string)

- 但假设参数还可能有函数对象怎么办?

- 例如

```
class ReadFromFile {
public:
    string operator()(){
        string input;
        getline(ifstream("input.txt"), input);
        return input;
    }
};
```

使用模板函数

```
#include <iostream>
#include <fstream>
using namespace std;

//省略readFromScreen/ReadFromFile/calculateAdd/writeToScreen

template<class ReadFunc, class CalFunc, class WriteFunc>
void process(ReadFunc read, CalFunc calculate, WriteFunc write)
{
    string data = read();
    string output = calculate(data);
    write(output);
}

int main()
{
    process(readFromScreen, calculateAdd, writeToScreen);
    process(ReadFromFile(), calculateAdd, writeToScreen);
    return 0;
}
```

完整代码上传到网络学堂

例子：一个简单计算器

- 想用数组储存选项，下面的代码ok吗？

```
auto readArr[] = {readFromScreen,
                  ReadFromFile()};
process(readArr[0], calculate, write);
process(readArr[1], calculate, write);
```

- auto是什么类型？
  - 无法推导！
  - 函数指针和函数对象不是同一种类型

- 需要一个类型能够统一两者

std::function类

- std::function类，来自<functional>头文件

```
template <class Ret, class... Args> class
function<Ret(Args...)>;
```

返回值　参数列表

```
function<string()> readArr[] =
          {readFromScreen, ReadFromFile()};
function<string(string)> calculateArr =
          {calculateAdd, CalculateMul()};
function<void(string)> writeArr[] =
          {writeToScreen, WriteToFile()};
```

- function为函数指针与对象提供了统一的接口

std::function类

```
#include <iostream>
#include <fstream>
#include <functional>
using namespace std;

//省略readFromScreen/ReadFromFile
int main()
{
    function<string()> readArr[] =
            {readFromScreen, ReadFromFile()};
    function<string()> readFunc;
    readFunc = readFromScreen; //允许函数的赋值
    readFunc = ReadFromFile();
    string (*readFunc2)();
    readFunc2 = readFromScreen;
    //readFunc2 = ReadFromFile(); //错误，类型不一致
    return 0;
}
```

使用function

```
#include <iostream>
#include <fstream>
#include <functional>
using namespace std;

//省略readFromScreen\ReadFromFile\calculateAdd\writeToScreen
void process(
        function<string()> read,
        function<string(string)> calculate,
        function<void(string)> write)
{
    string data = read();
    string output = calculate(data);
    write(output);
}
int main()
{
    process(readFromScreen, calculateAdd, writeToScreen);
    process(ReadFromFile(), calculateAdd, writeToScreen);
    return 0;
}
```

对比几种实现方式

- 使用虚函数实现
  - 需要构造基类和子类
  - 晚绑定（运行时绑定）
- 使用模板实现
  - 可以支持函数指针和函数对象
    （通过模板，自动重载实现）
  - 早绑定（编译期绑定）
- 使用std::function实现
  - 也可以支持函数指针和函数对象
    （通过function的多态）
  - 晚绑定（运行时绑定）

std::function的意义

- 函数对象化
  - 万物皆对象，符合OOP的设计理念
  - 函数可以作为参数传递
  - 函数可以作为变量储存

- 解决Duck Typing的繁琐问题
  - 不再需要模板来调用不同的函数
  - 简化理解，所有的函数都可以看做std::function

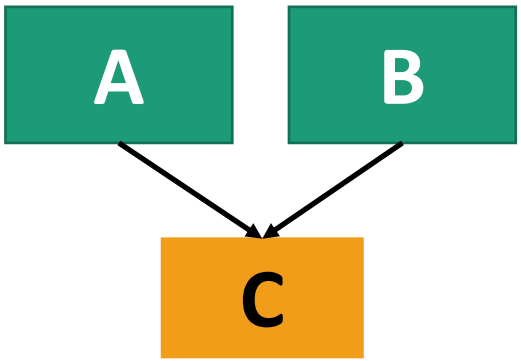
STL与函数对象

- STL有大量函数用到了函数对象(#include <algorithm>)
  - for_each　　对序列进行指定操作
  - find_if　　找到满足条件的对象
  - count_if　　对满足条件的对象计数
  - binary_search　　二分查找满足条件的对象
  - ……
- 并且也有许多预置的函数对象(#include <functional>)
  - less　　比较a<b
  - equal_to　　比较a==b
  - greater　　比较a>b
  - plus　　返回a+b
  - ……
- 熟练使用函数对象有助于实现复杂的功能

智能指针与引用计数

指针的销毁

- 一个例子：
  - A、B对象共享一个C对象
  - C对象不想交由外部销毁
- A、B中的谁负责销毁C？



- 应该在A、B都销毁时C才能销毁
- 如何自动的处理？

智能指针

- shared_ptr 来自<memory>库
  - 构造方法

```
shared_ptr<int> p1(new int(1));
shared_ptr<MyClass> p2 = make_shared<MyClass>(2);
shared_ptr<MyClass> p3 = p2;
shared_ptr<int> p4;   //空指针
```

  - 访问对象

```
int x = *p1;       //从指针访问对象
int y = p2->val; //访问成员变量
```

  - 销毁对象

    p2和p3指向同一对象，当两者均出作用域才会被销毁

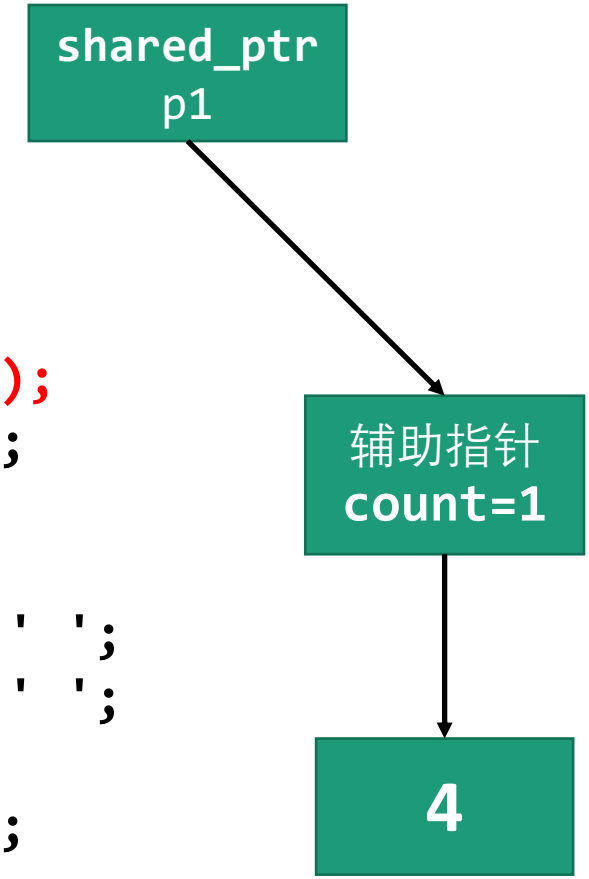
引用计数

- 为什么智能指针能够知道何时销毁对象?
- 引用计数！当引用计数归0时，销毁对象

```
#include <memory>
#include <iostream>
using namespace std;
int main()
{
    shared_ptr<int> p1(new int(4));
    cout << p1.use_count() << ' '; // 1
    {
        shared_ptr<int> p2 = p1;
        cout << p1.use_count() << ' '; // 2
        cout << p2.use_count() << ' '; // 2
    }  //p2出作用域
    cout << p1.use_count() << ' '; // 1
}
```

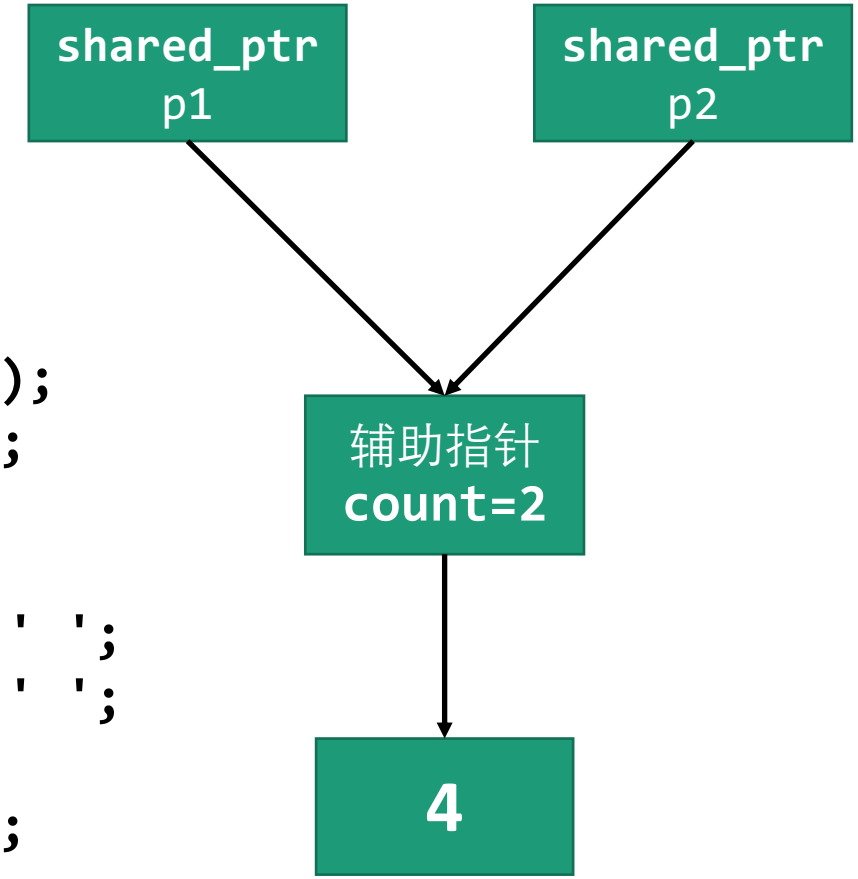
运行过程

```
{
shared_ptr<int> p1(new int(4));
cout << p1.use_count() << ' ';
{
    shared_ptr<int> p2 = p1;
    cout << p1.use_count() << ' ';
    cout << p2.use_count() << ' ';
}   //p2出作用域
cout << p1.use_count() << ' ‘;
}
```



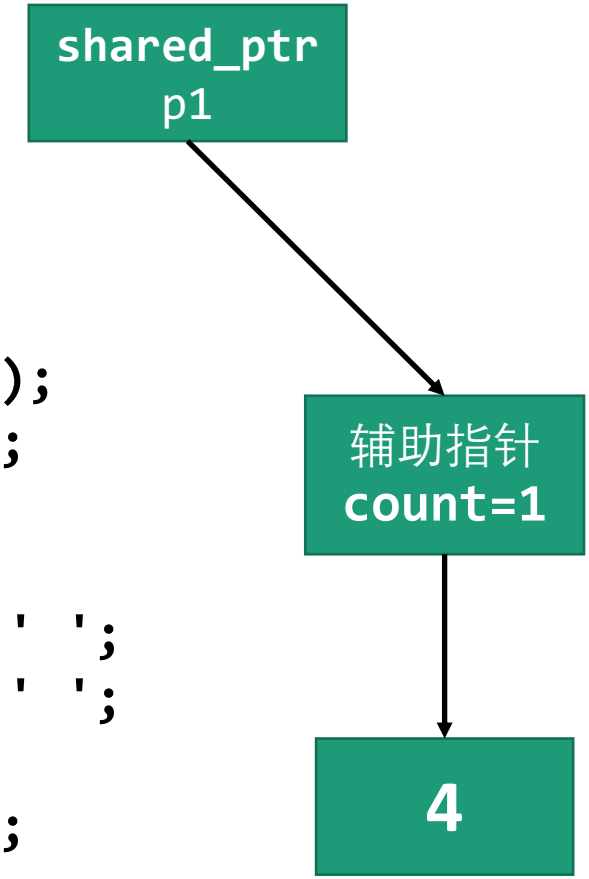
运行过程

```
{
shared_ptr<int> p1(new int(4));
cout << p1.use_count() << ' ';
{
    shared_ptr<int> p2 = p1;
    cout << p1.use_count() << ' ';
    cout << p2.use_count() << ' ';
}   //p2出作用域
cout << p1.use_count() << ' ‘;
}
```



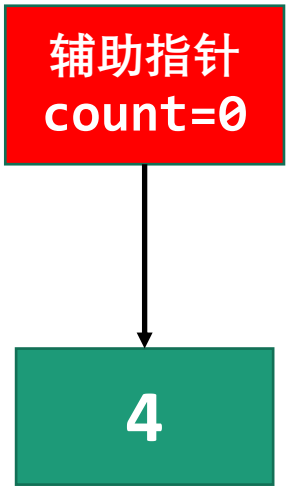
运行过程

```
{
shared_ptr<int> p1(new int(4));
cout << p1.use_count() << ' ';
{
    shared_ptr<int> p2 = p1;
    cout << p1.use_count() << ' ';
    cout << p2.use_count() << ' ';
}   //p2出作用域
cout << p1.use_count() << ' ‘;
}
```



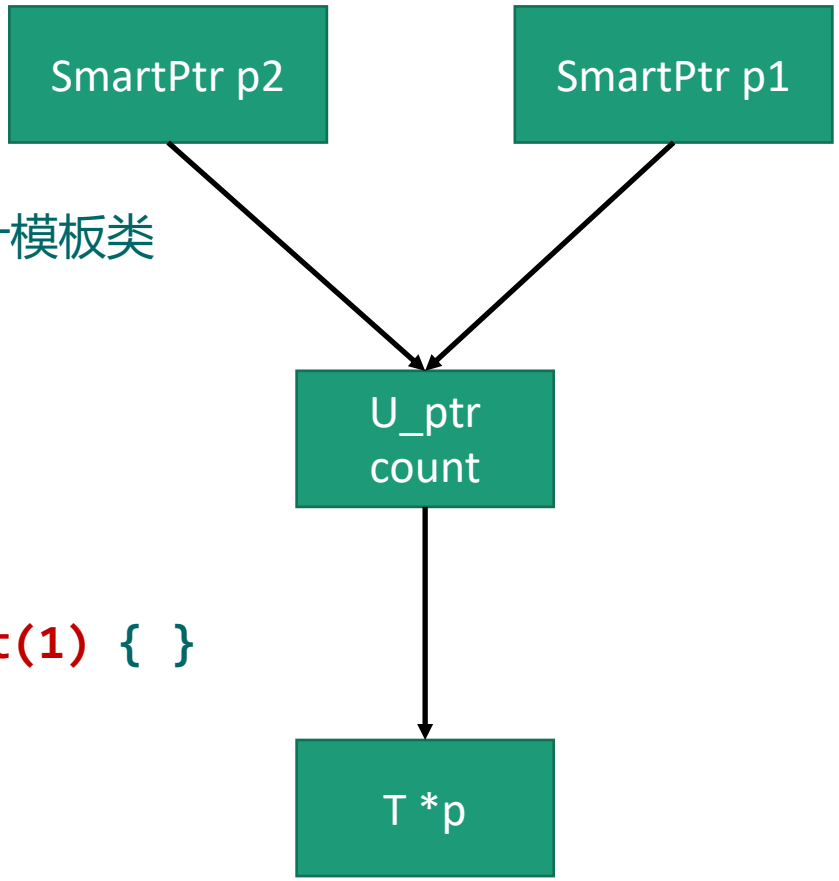
运行过程

```
{
shared_ptr<int> p1(new int(4));
cout << p1.use_count() << ' ';
{
    shared_ptr<int> p2 = p1;
    cout << p1.use_count() << ' ';
    cout << p2.use_count() << ' ';
}   //p2出作用域
cout << p1.use_count() << ' ‘;
}
//调用delete，销毁int*
```



实现自己的引用计数

```
#include <iostream>
using namespace std;

template <typename T>
class SmartPtr;  //声明智能指针模板类

template <typename T>
class U_Ptr { //辅助指针
private:
  friend class SmartPtr<T>;
  //SmartPtr是U_Ptr的友元类
  U_Ptr(T *ptr) :p(ptr), count(1) { }
  ~U_Ptr() { delete p; }

  int count;
  T *p; //实际数据存放
};
```



实现自己的引用计数

```
template <typename T>
class SmartPtr { //智能指针
  U_Ptr<T> *rp;
public:
  SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }
  SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) {
    ++rp->count;
  }
  SmartPtr& operator=(const SmartPtr<T>& rhs) {
    ++rhs.rp->count;
    if (--rp->count == 0) //减少自身所指rp的引用计数 pA = pB
      delete rp; //删除所指向的辅助指针
    rp = rhs.rp;
    return *this;
  }
  ~SmartPtr() {
    if (--rp->count == 0)
      delete rp;
  }
```

实现自己的引用计数

```
  T & operator *() { return *(rp->p); }
  T* operator ->() { return rp->p; }
};

int main(int argc, char *argv[]) {
  int *pi = new int(2);
  SmartPtr<int> ptr1(pi); //构造函数
  SmartPtr<int> ptr2(ptr1); //拷贝构造
  SmartPtr<int> ptr3(new int(3)); //能否ptr3(pi)???
  ptr3 =  ptr2; //注意赋值运算
  cout << *ptr1 << endl; //输出2
  *ptr1 = 20;
  cout << *ptr2 << endl; //输出20

  return 0;
}
```

shared_ptr的其他用法

- 其他用法
  - p.get()　　获取裸指针
  - p.reset()　　清除指针并减少引用计数
  - static_pointer_cast<int>(p)
  - dynamic_pointer_cast<Base>(p)
- 注意!
  - 不能使用同一裸指针初始化多个智能指针

```
int* p = new int();
shared_ptr<int> p1(p); shared_ptr<int> p2(p);
                       // 会产生多个辅助指针!
```

  - 不能直接使用智能指针维护对象数组: Object* p = new Object[10];

    (为什么?)

智能指针不总是智能

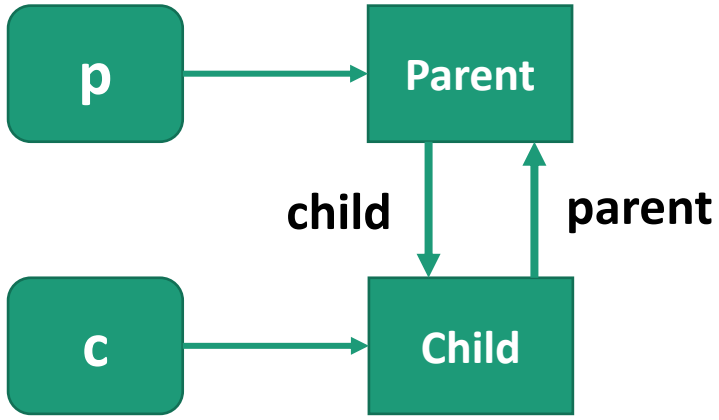
- Parent类和Child类

```
#include <memory>
#include <iostream>
using namespace std;

class Child;
class Parent {
    shared_ptr<Child> child;
public:
    Parent() {cout << "parent constructing" << endl; }
    ~Parent() {cout << "parent destructing" << endl; }
    void setChild(shared_ptr<Child> c) {
        child = c;
    }
};
class Child {
    shared_ptr<Parent> parent;
public:
    Child() {cout << "child constructing" << endl; }
    ~Child() {cout << "child destructing" << endl; }
    void setParent(shared_ptr<Parent> p) {
        parent = p;
    }
};
```

智能指针不总是智能

- Parent类和Child类

```
void test() {
    shared_ptr<Parent> p(new Parent());
    shared_ptr<Child> c(new Child());
    p->setChild(c);
    c->setParent(p);
    //p和c被销毁
}

int main()
{
    test();
    return 0;
}
```



输出结果：

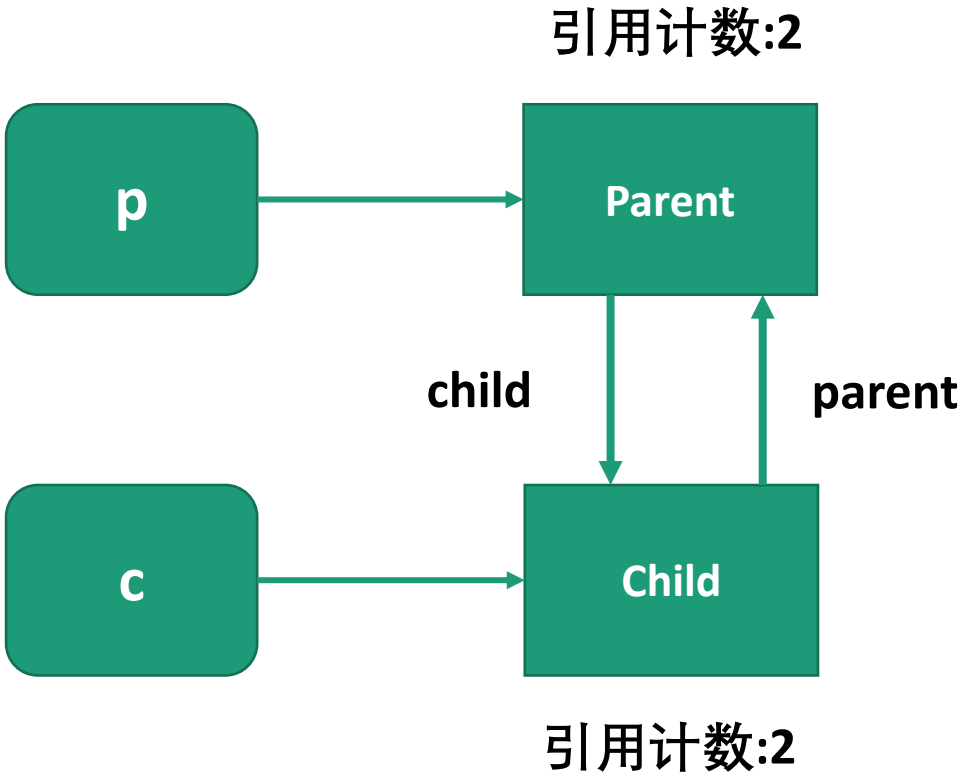
parent constructing

child constructing

没有析构？

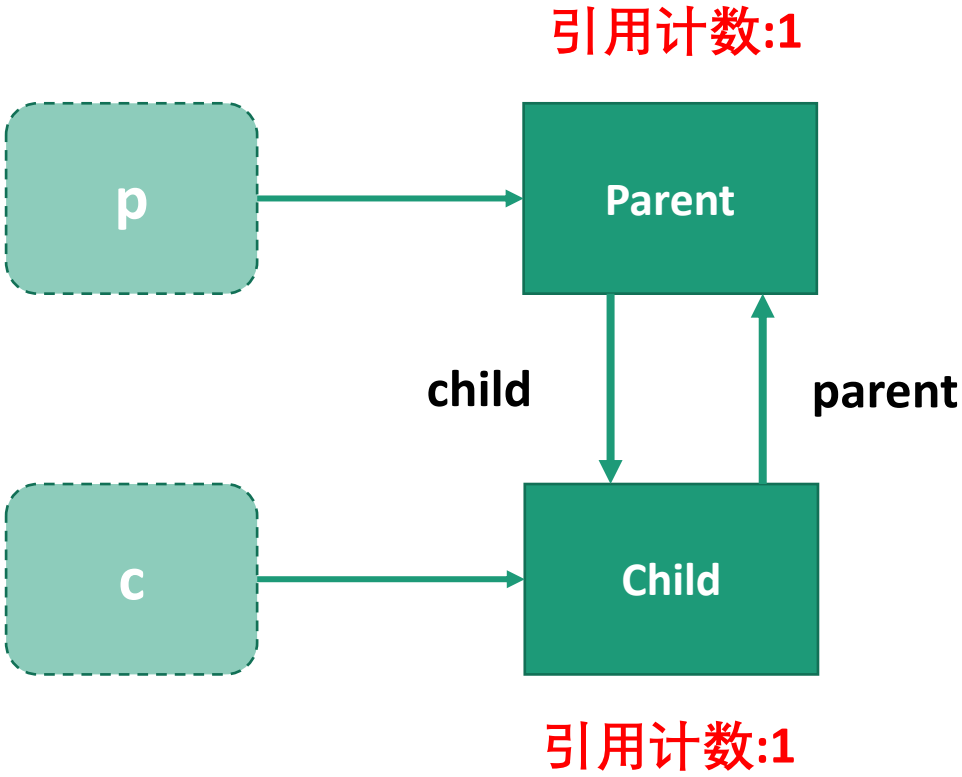
智能指针不总是智能

- 为什么？



智能指针不总是智能

- 为什么？



- 两个对象的引用次数都是1，内存泄漏！

智能指针不总是智能

- 修改Child类

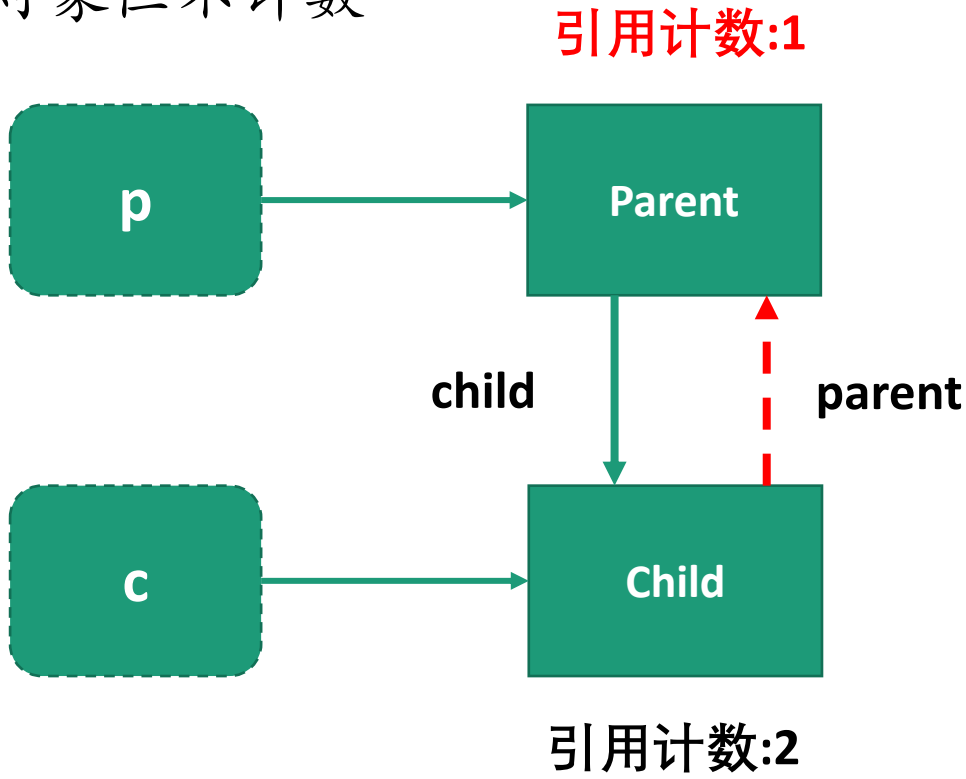
```
class Child {
    weak_ptr<Parent> parent;  //修改为弱引用
public:
    Child() {cout << "child constructing" << endl; }
    ~Child() {cout << "child destructing" << endl; }
    void setParent(shared_ptr<Parent> p) {
        parent = p;
    }
};
```

输出结果：

parent constructing
child constructing
parent destructing
child destructing

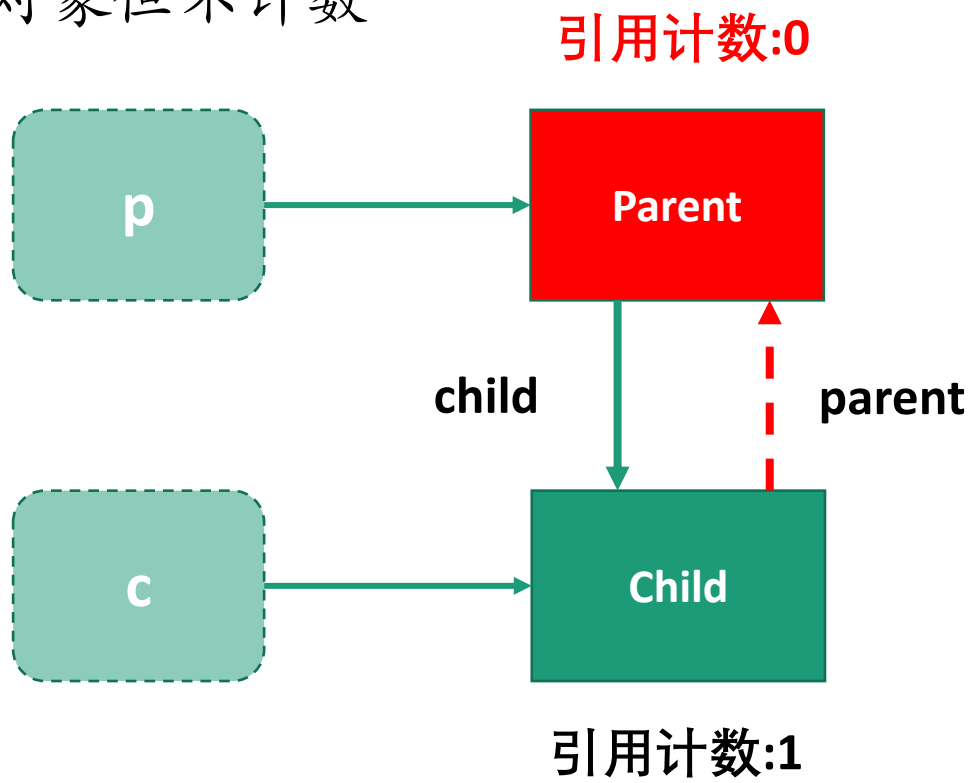
智能指针不总是智能

- 弱引用weak_ptr
  - 指向对象但不计数



智能指针不总是智能

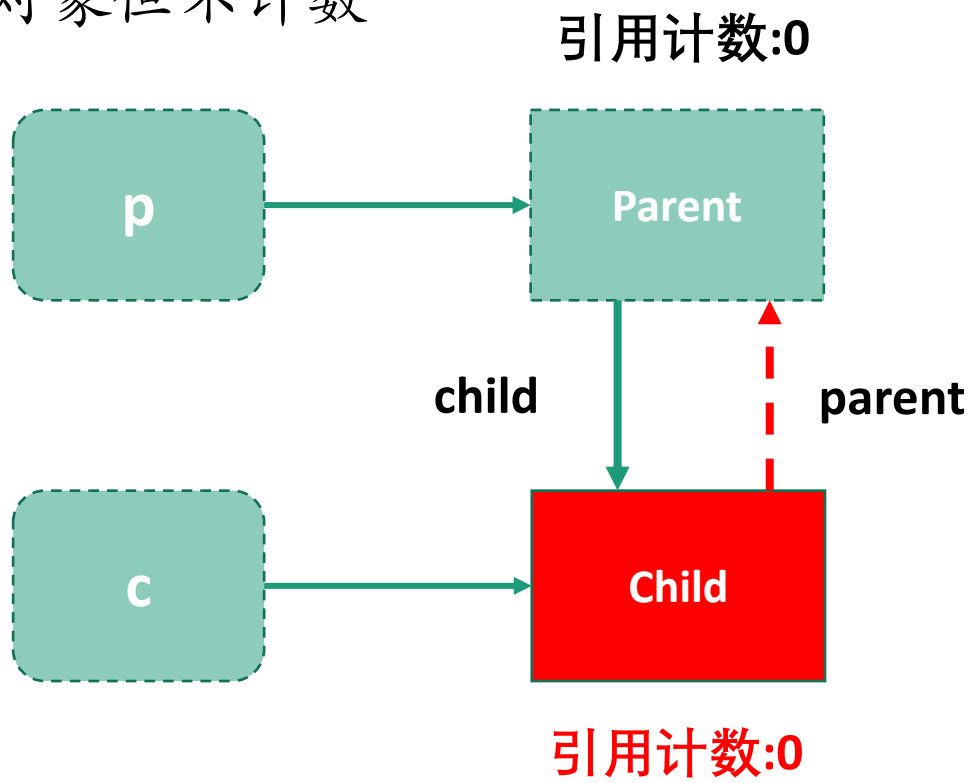
- 弱引用weak_ptr
  - 指向对象但不计数



  - Parent即将销毁

智能指针不总是智能

- 弱引用weak_ptr
  - 指向对象但不计数



  - Child即将销毁

弱引用

- 弱引用指针的创建

```
shared_ptr<int> sp(new int(3));
weak_ptr<int> wp = sp;
```

- 弱引用指针的用法

```
wp.use_count()   //获取引用计数
wp.reset()       //清除指针
wp.expired()     //检查对象是否无效
sp = wp.lock()   //从弱引用获得一个智能指针
```

例子：弱引用

```
#include <memory>
#include <iostream>
using namespace std;
int main()
{
    std::weak_ptr<int> wp;
    {
        auto sp1 = std::make_shared<int>(20);
        wp = sp1;
        cout << wp.use_count() << endl;  //1
        auto sp2 = wp.lock();  //从弱引用中获得一个shared_ptr
        cout << wp.use_count() << endl;  //2
        sp1.reset();                     //sp1释放指针
        cout << wp.use_count() << endl;  //1
    }  //sp2销毁
    cout << wp.use_count() << endl;  //0
    cout << wp.expired() << endl;   //检查弱引用是否失效：True
    return 0;
}
```

独享所有权

- shared_ptr涉及引用计数，性能较差
- 如果要保证一个对象只被一个指针引用
- unique_ptr

```
#include <memory>
#include <utility>
using namespace std;
int main() {
    auto up1 = std::make_unique<int>(20);
    //unique_ptr<int> up2 = up1;
                //错误，不能复制unique指针
    unique_ptr<int> up2 = std::move(up1);
                //可以移动unique指针
    int* p = up2.release();
                //放弃指针控制权，返回裸指针
    delete p;
    return 0;

}
```

智能指针总结

- 优点
  - 智能指针可以帮助管理内存，避免内存泄露
  - 区分unique_ptr和shared_ptr能够明确语义
  - 在手动维护指针不可行，复制对象开销太大时，智能指针是唯一选择。
- 缺点
  - 引用计数会影响性能
  - 智能指针不总是智能，需要了解内部原理
  - 需要小心环状结构和数组指针

拓展阅读

- 智能指针来自于<memory>库，负责对动态内存管理的封装
  - http://www.cplusplus.com/reference/memory/
- 主要包含四个部分
  - Allocators　内存创建
  - Manage Pointers　智能指针，本节内容
    还包含一些辅助智能指针使用的函数
  - Uninitialized Memory　对未初始化内存的操作
  - Memory Model　其他内存管理

拓展自学

- 以下哪些能正常编译?

```
int func1(const int &x, int &b) {...};
function<______> pf1 = func1;
```

A. int(int, int)
B. int(int, int&)
C. int(int&, int&)
D. int(const int&, int&)
E. int(const int, int)
F. int&(int, int&)

拓展自学

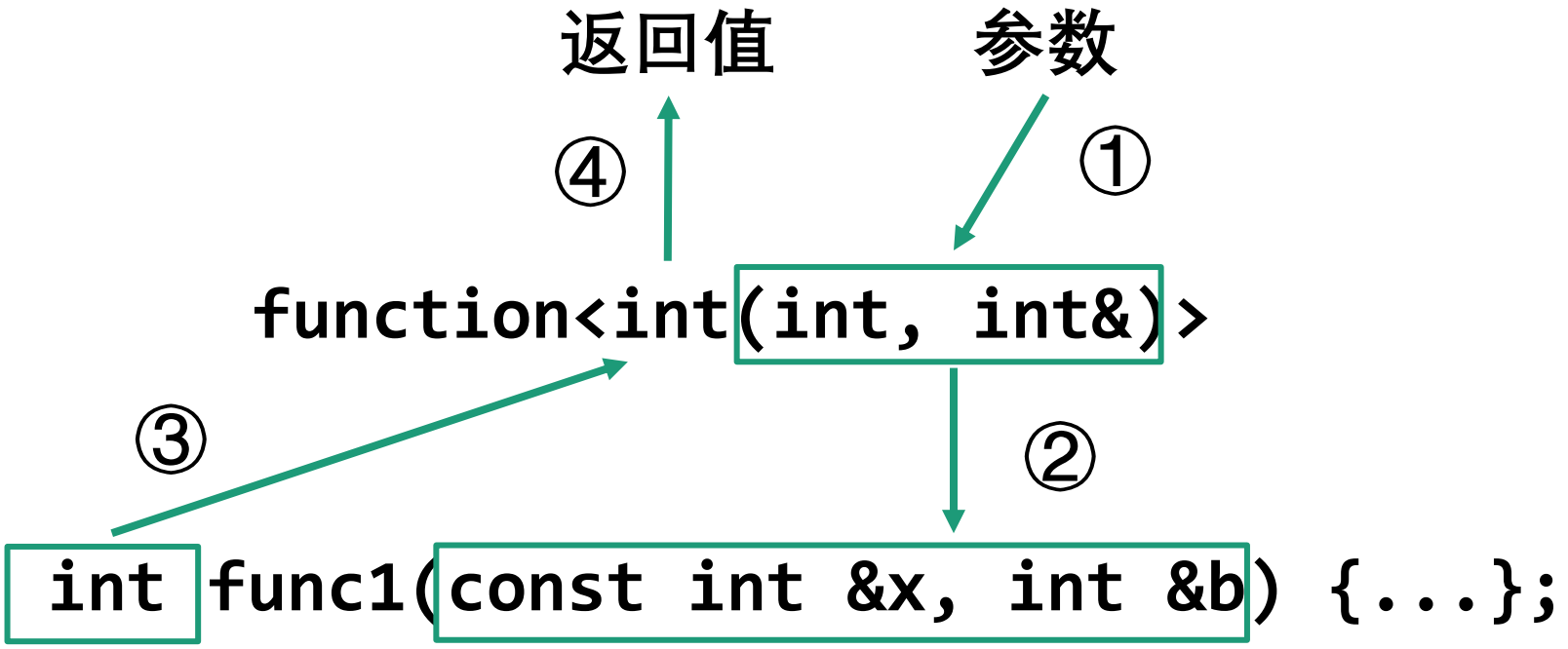
- 以下哪些能正常编译?

```
int func1(const int &x, int &b) {...};
function<______> pf1 = func1;
```

A. int(int, int)
B. int(int, int&)
C. int(int&, int&)
D. int(const int&, int&)
E. int(const int, int)
F. int&(int, int&)

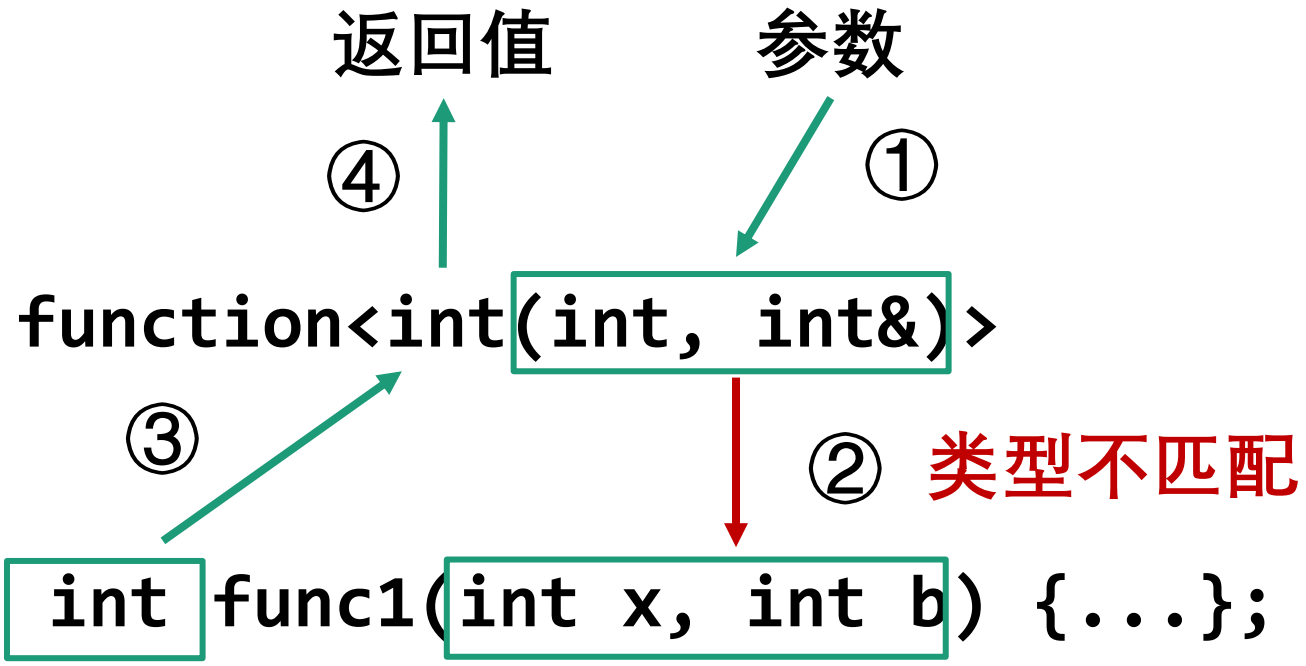
拓展自学

- 考虑调用时的顺序



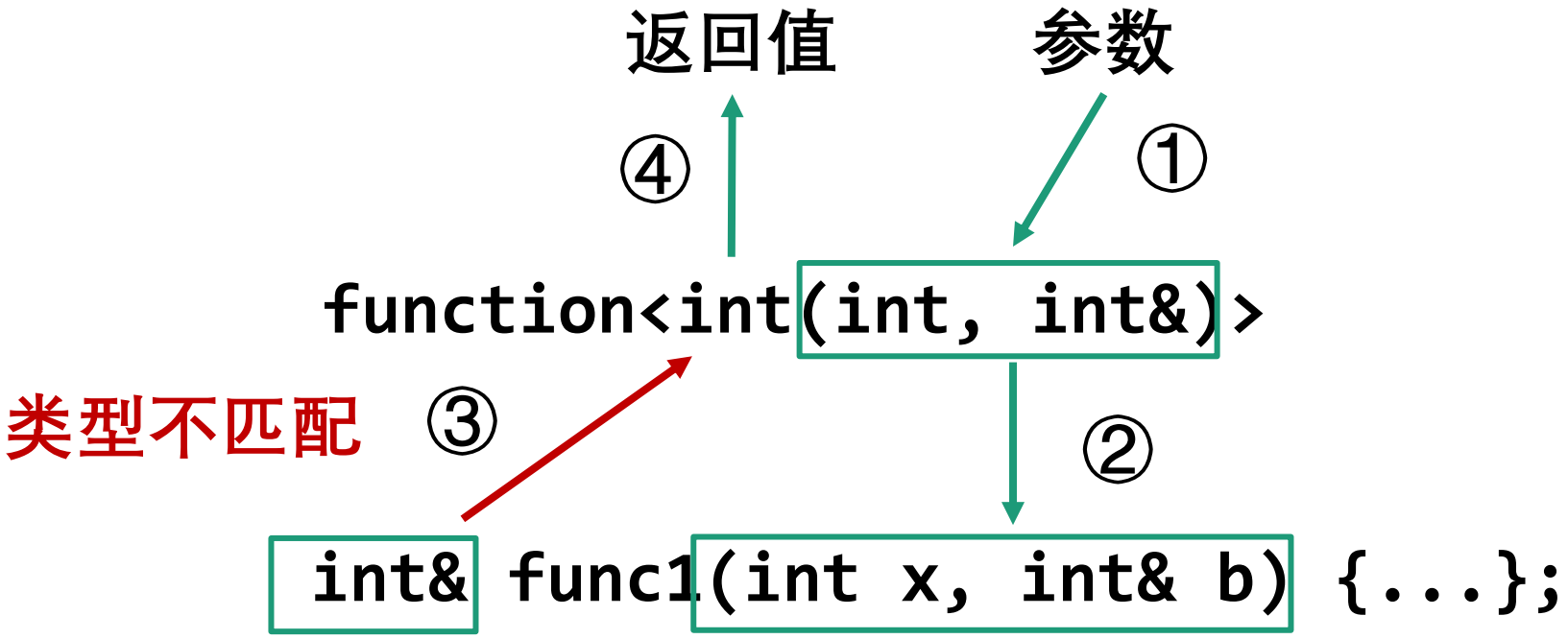
拓展自学

- 考虑调用时的顺序



拓展自学

- 考虑调用时的顺序



拓展自学

- 以下哪些能正常编译?

```
class Func2{
public:
   int& operator()(int &&b) const {...}
};
function<______> pf2 = func2;
```

A. int&(int&&)
B. int(int&&)
C. int&(int&)
D. int(int&)

拓展自学

- 以下哪些能正常编译?

```
class Func2{
public:
   int& operator()(int &&b) const {...}
};
function<_____> pf2 = func2;
```

A. int&(int&&)
B. int(int&&)
C. int&(int&)
D. int(int&)

请自己尝试：
int&(int)是否可以？
为什么？

结束